Corso di Compilatori
Anno accademico
2015/2016

LEXER
{go}

Supervisore
Prof. Gennaro Costagliola

Studenti
Luigi Lomasto
Andrea Soldà

Obiettivi

L’obiettivo è quello di creare un Lexer in grado di fare l’analisi lessicale di un qualsiasi file GO. Di seguito sarà illustrata la fase implementative e verranno mostrati i file GO analizzati.

Realizzazione del Lexer

Per l'implementazione del Lexer è stato utilizzato il tool Jflex, un generatore realizzato in Java che, a partire dalle specifiche lessicali, ne genera un analizzatore lessicale.

Per prima cosa, abbiamo studiato il manuale di riferimento del linguaggio GO, che è possibile visionare al link sottostante.

https://golang.org/ref/spec#Integer_literals

Ci siamo focalizzati quindi sugli elementi lessicali del linguaggio elencati di seguito.

- Comments
- Tokens
- Semicolons
- Identifiers
- Keywords
- Operators and Delimiters
- Integer literals
- Floating-point literals
- Imaginary literals
- Rune literals
- String literals

Per poter usare i simboli della classe *sym.java* abbiamo importato la libreria **java_cup.runtime**. Di solito tale file viene generato automaticamente dal file .cup a partire dalla grammatica del linguaggio. Per poter memorizzare le

tabelle dei simboli di stringhe e simboli, è stata importata la classe **java.util.Hashtable** e realizzata una classe (Tables.java) contenente le rispettive tabelle di stringhe e simboli. Inoltre, sono state utilizzate le seguenti direttive Jflex:

- **%class Lexer :** Per assegnare un nome alla classe.
- **%unicode :** Per la compatibilità tra caratteri.
- **%cup :** Necessaria per interfacciarsi con un parser CUP
- **%line :** Conteggio automatico della linea dell'input
- **%column :** Conteggio automatico della colonna dell'input
- **%debug:** Simula il comportamento del Lexer con la stampa delle azioni.

Gli stati utilizzati sono sostanzialmente 2:

- **String :** Gestisce le stringhe.
- **Comment** : Gestisce i commenti.

Definizioni regolari, keywords, operatori

Definizioni regolari

- Letter = [a-zA-Z_]
- Decimal_digit = [0-9]
- Octal_digit = [0-7]
- Hex_digit = [0-9a-fA-F]
- Hex_lit = "0" [xX] [0-9a-fA-F] {Hex_digit}*
- Decimal_lit = [1-9] {Decimal_digit}* | "0"
- Octal_lit = "0" {Octal_digit}*
- Float_lit = ({Decimal_lit} "." {Decimal_lit})
- Pointer = (*{Identifier})
- Imaginary_lit = {Decimal_lit } "i" | {Float_lit} "i"
- Identifier = [a-zA-Z_] {Letter}*
- Comment = {TraditionalComment} | {EndOfLineComment} | {DocumentationComment}
- TraditionalComment = "/*" [^*] ~"*/" | "/*" "*"+ "/"
- EndOfLineComment = "//" {InputCharacter}* {LineTerminator}?
- DocumentationComment = "/**" {CommentContent} "*"+ "/"
- CommentContent = ([^*] | *+ [^/*])*

Keywords

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Operatori

```
+    &     +=    &=     &&    ==    !=    (    )
-    |     -=    |=     ||    <     <=    [    ]
*    ^     *=    ^=     <-    >     >=    {    }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     ...   .    :
     &^          &^=
```

Test

La classe di test è costituita da un unico metodo: il main. All'interno del main vengono dichiarati gli oggetti Symbol e Lexer. L'oggetto Lexer, prende in input il nome del file da analizzare e mediante l'oggetto Symbol stampa il flusso di token trovati.

Risultati

I test sono stati fatti su tre file, un file good.go (file primo di errori), un file bad.go (file con errori) ed un file raw_tokens.go (contenente tutte le keywords). Alla fine della documentazione è riportato per ogni token numerico, il rispettivo nome.

Good.go

```
/* Commento di prova */
package main
// commento alla funzione
func main() {
    i := 03
    a := 0
      int variabile
    float32 floatNumber 10.2
    10i
```

```
    String stringa := "waw"
    for i < 100 {
        a += i
    }
    if a < 100 {
        a = 0xFF
    } else {
        a *= 20
    }
}
```

Flusso di token

```
< 543 > at line 1
< 80 > at line 2
< 5 > at line 2
< 543 > at line 3
< 70 > at line 4
< 5 > at line 4
< 13 > at line 4
< 14 > at line 4
< 21 > at line 4
< 5 > at line 5
< 52 > at line 5
< 7 > at line 5
< 5 > at line 6
< 52 > at line 6
< 6 > at line 6
< 100 > at line 7
< 5 > at line 7
< 130 > at line 8
< 5 > at line 8
< 876 > at line 8
< 678 > at line 9
< 5 > at line 10
< 5 > at line 10
< 52 > at line 10
buffer: waw
< 28 > at line 10
< -1 > at line 10
< 59 > at line 11
< 5 > at line 11
< 16 > at line 11
< 6 > at line 11
< 21 > at line 11
< 5 > at line 12
< 37 > at line 12
< 5 > at line 12
< 22 > at line 13
< 34 > at line 14
< 5 > at line 14
< 16 > at line 14
< 6 > at line 14
< 21 > at line 14
< 5 > at line 15
< 8 > at line 15
< 58 > at line 15
< 22 > at line 16
< 69 > at line 16
< 21 > at line 16
< 5 > at line 17
< 33 > at line 17
< 6 > at line 17
< 22 > at line 18
< 22 > at line 19
```

In questo test, il flusso di token generato rispecchia il codice scritto.

Bad.go

```
/pakage main
/* commento su
piu linee */
func mmain6() {
    var zz inte
    fori := 0; i < 10; i+++ {
        _nosense(i)
    }
    String stringa = "Stringa senza chiusura

}
```

Flusso di token

```
< 3 > at line 1
< 5 > at line 1
< 5 > at line 1
< 543 > at line 2
< 70 > at line 4
< 5 > at line 4
< 6 > at line 4
< 13 > at line 4
< 14 > at line 4
< 21 > at line 4
< 61 > at line 5
< 5 > at line 5
< 5 > at line 5
< 5 > at line 6
< 52 > at line 6
< 6 > at line 6
< 25 > at line 6
< 5 > at line 6
< 16 > at line 6
< 6 > at line 6
< 25 > at line 6
< 5 > at line 6
< 38 > at line 6
< 9 > at line 6
< 21 > at line 6
< 5 > at line 7
< 13 > at line 7
< 5 > at line 7
< 14 > at line 7
< 22 > at line 8
< 5 > at line 9
< 5 > at line 9
< 8 > at line 9
< -1 > at line 9
< 22 > at line 11
```

Alla linea 1 , “/pakage main”, non viene riconosciuto come commento, ma come una divisione più due identificatori. Alla riga 9 , in corrispondenza della stringa non chiusa, viene restituito un token di “*ERROR STRING”*.

Row_tokens.go

In questo file sono state inserite numerose keywords del linguaggio per verificare il corretto riconoscimento da parte del lexer.

```
break
default
func
interface
select
case
defer
go
map
struct
chan
else
goto
package
switch
const
if
range
type
continue
for
import
return
var
"una stringa"
//un commento su singola linea
/*un commento su
due linee */
123
0xA
wdwqd
int sadsadsadsadasd
||
```

Flusso di token

```
< 5 > at line 1
< 81 > at line 2
< 70 > at line 3
< 83 > at line 4
< 78 > at line 5
< 66 > at line 6
< 75 > at line 7
< 35 > at line 8
< 60 > at line 9
< 4 > at line 10
< 67 > at line 11
< 69 > at line 12
< 68 > at line 13
< 80 > at line 14
< 79 > at line 15
< 72 > at line 16
< 34 > at line 17
< 74 > at line 18
```

```
< 71 > at line 19
< 82 > at line 20
< 59 > at line 21
< 77 > at line 22
< 76 > at line 23
< 61 > at line 24
< 28 > at line 25
< -1 > at line 25
< 543 > at line 26
< 543 > at line 27
< 6 > at line 29
< 58 > at line 30
< 5 > at line 31
< 100 > at line 32
< 5 > at line 32
< 46 > at line 33
```

Lista di token con rispettivo valore numerico

```
EOF = 0;
DIV = 3;
TIME = 4;
IDENTIFIER = 5;
DECIMAL_LITERAL = 6;
OCTAL_LITERAL = 7;
EQ = 8;
PLUS = 9;
MINUS = 10;
AND = 11;
NOT = 12;
RO = 13;
RC = 14;
OR = 15;
AO = 16;
OS = 17;
CS = 18;
CAP = 19;
AC = 20;
BRACEO = 21;
BRACEC = 22;
TP = 23;
COMMA = 24;
SEMICOLON = 25;
PERC = 26;
POINT = 27;
STRING = 28;
STRING_LITERAL = 30;
DIVEQ = 32;
TIMEEQ = 33;
IF = 34;
GO = 35;
EQEQ = 36;
PLUSEQ = 37;
PLUSPLUS = 38;
MINUSEQ = 39;
MINUSMINUS = 40;
ANDEQ = 41;
ANDAND = 42;
ANDCAP = 43;
NOTEQ = 44;
OREQ = 45;
OROR = 46;
AOMINUS = 47;
```

```
AOAO = 48;
CAPEQ = 49;
AOEQUALS = 50;
ACAC = 51;
TPEQ = 52;
PERCEQ = 53;
HEX_LITERAL = 58;
FOR = 59;
MAP = 60;
VAR = 61;
ANDCAPEQ = 62;
AOAOEQ = 63;
ACACEQ = 64;
POINTPOINTPOINT = 65;
CASE = 66;
CHAN = 67;
GOTO = 68;
ELSE = 69;
FUNC = 70;
TYPE = 71;
CONST = 72;
BREAK= 73;
RANGE = 74;
DEFER = 75;
RETURN = 76;
IMPORT = 77;
SELECT = 78;
SWITCH = 79;
PACKAGE = 80;
DEFAULT = 81;
CONTINUE = 82;
INTERFACE = 83;
FALLTHROUGH = 84;
INT = 100;
BYTE = 200;
RUNE = 300;
UINT = 400;
INT8 = 500;
INT16 = 600;
INT64 = 700;
INT32 = 800;
UINT8 = 900;
UINT64 = 1000;
UINT32 = 110;
FLOAT64 = 120;
FLOAT32 = 130;
UINTPTR = 123;
COMPLEX64 = 345;
UINT16 = 34645;
COMPLEX128 = 546;
COMMENT = 543;
IMAGINARY_LITERAL=678;
FLOAT_LITERAL=876;
ERROR_STRING = -1;
SINGLEQUOTE = -10;
ERROR = -20;
```